COMPUTER VISION, GRAPHICS, AND IMAGE PROCESSING **26**, 1–16 (1984)

Algorithms for the Conversion of Quadtrees to Rasters

HANAN SAMET

Computer Science Department, University of Maryland, College Park, Maryland 20742

Received November 24, 1980; accepted December 2, 1982

A number of algorithms are presented for obtaining a raster representation for an image given its quadtree. The algorithms are given in an evolutionary manner starting with the straightforward top-down approach that visits each run in a row in succession starting at the root of the tree. The remaining algorithms proceed in a manner akin to an inorder tree traversal. All of the algorithms are analyzed and an indication is given as to when each is preferable. The execution time of all of the algorithms is shown to be proportional to the sum of the heights of the blocks comprising the image.

1. INTRODUCTION

Region representation is an important issue in applications such as image processing, computer graphics, and cartography (see [2] for a brief review of representations currently in use). Recently there has been much interest in the quadtree [2, 4–10], a compact hierarchical representation which, depending on the nature of the image, saves space as well as facilitates operations such as search. A number of algorithms have been developed for conversion between the quadtree and other representations such as boundary codes [2] as well as the construction of a quadtree given such representations [13–15]. In this paper we present and compare a number of algorithms for obtaining a raster representation of an image given its quadtree. The importance of the algorithms described here is that they enable raster-like display devices to sequentially output (i.e., a row at a time) an image defined by a quadtree without requiring the storage of the matrix corresponding to the image. Our algorithms are seen to require storage for at most the quadtree and one row of pixels. The row of pixels corresponds to the raster device's buffer.

We describe four algorithms in an evolutionary manner. All of the algorithms visit each row in sequence. The first algorithm is a straightforward top-down approach that visits each row independently. The remaining algorithms develop a bottom-up approach that visits the various segments of each row in a manner analogous to an inorder tree traversal [11]. The analysis shows that each algorithm has its advantages. In particular, the second, third, and fourth algorithms achieve time bounds which are lower when the resolution of the image increases.

In this and the next section we briefly define the representations used. Sections 3 and 4 contain the algorithms and an analysis of their execution times. We also include a formal presentation of the algorithms using a variant of ALGOL 60 [12] as well as motivations for their various steps.

We assume that the given image is a 2^n by 2^n array of unit square "pixels." The quadtree is an approach to image representation based on successive subdivision of

The support of the Defense Advanced Research Projects Agency and the U.S. Army Night Vision Laboratory under Contract DAAG-53-76C-0138 (DARPA Order 3206) is gratefully acknowledged, as is the help of Sally Atkinson in preparing this paper. The author has benefited greatly from discussions with Charles R. Dyer, Azriel Rosenfeld, and Robert E. Webber.

0734-189X/84 $3.00

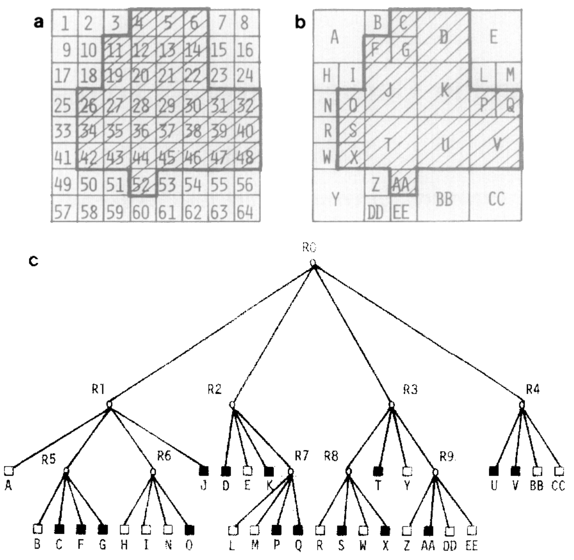

FIG. 1. An image, its maximal blocks, and the corresponding quadtree. Blocks in the image are shaded. (a) Sample image. (b) Block decomposition of the image in (a). (c) Quadtree representation of the blocks in (b).

the array into quadrants. In essence, we repeatedly subdivide the array into quadrants, subquadrants,..., until we obtain blocks (possibly single pixels) which consist entirely of 1's or 0's. This process is represented by a tree of out-degree 4 in which the root node represents the entire array, the four sons of the root node represent the quadrants, and the terminal nodes correspond to those blocks of the array for which no further subdivision is necessary. For example, Fig. 1b is a block decomposition of the region in Fig. 1a while Fig. 1c is the corresponding quadtree. In general, BLACK and WHITE square nodes represent blocks consisting entirely of 1's and 0's respectively. Circular nodes, also termed GRAY nodes, denote nonterminal nodes. Note that the quadtree representation discussed here should not be confused with the quadtree representation of two-dimensional point space data [3].

2. DEFINITIONS AND NOTATION

Let each node in a quadtree be stored as a record containing six fields. The first five fields contain pointers to the node's father and its four sons, i.e., quadrants, labeled NW, NE, SE, and SW. Given a node P and a son I, these fields are referenced as FATHER(P) and SON(P, I) respectively. We assume that the FATHER of the root of the tree is NULL. We can determine the specific quadrant in which a node, say P, lies relative to its father by use of the function SONTYPE(P) which has a value of I if SON(FATHER(P), I) = P. The sixth field, named NODETYPE, describes the contents of the block of the image which the node represents—i.e., BLACK, WHITE, or GRAY.

The four sides of a node's block are called its N, E, S, and W sides. They are also termed its boundaries and at times we speak of them as if they were directions. Figure 2 shows the relationship between the quadrants of a node's block and its boundaries. A node, say Q, is said to be a *neighbor* of another node, say P, in

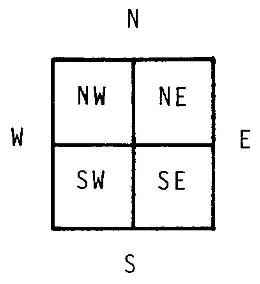

FIG. 2. Relationship between a block's four quadrants and its boundaries.

direction D if Q corresponds to the smallest block having a side that is adjacent to all of side D of P's block (not just at a corner). For example, in Fig. 1, BLACK node J is the neighbor of BLACK node F in direction S; similarly, GRAY node R is the S neighbor of node A.

In order to determine a node's neighbor in a specified direction, we have to traverse father links until a common ancestor of the two nodes is found. Once the common ancestor is located, we descend along a path that retraces the previous path with the modification that each step is a reflection of the corresponding step with respect to the axis formed by the common boundary between the two nodes. For example, when attempting to locate the S neighbor of node P in Fig. 1 (i.e., node V), node $R0$ is the common ancestor, and the S edge of the block corresponding to node P is the common boundary. During this process we pass through nodes $R7$, $R2$, $R0$, $R4$, and V in order. The encoding of this procedure is facilitated by the following predicates and functions. ADJ(B, I) is true if and only if quadrant I is adjacent to boundary B of the node's block; e.g., ADJ(N, NW) is true. REFLECT (B, I) yields the SONTYPE value of the block of equal size that is adjacent to side B of a block having SONTYPE value I. For example: REFLECT(W, NW) = NE, REFLECT(E, NW) = NE, REFLECT(N, NW) = SW, and REFLECT(S, NW) = SW.

Given a quadtree corresponding to a 2^n by 2^n array, we say that the root node is at level n, and that a node at level i is at a distance of $n - i$ from the root of the tree. In other words, for a node at level i, we must ascend $n - i$ FATHER links to reach the root of the tree. Note that the farthest node from the root of the tree is at level ≥ 0. A node at level 0 corresponds to a single pixel in the image. Also, we say that a node is of size 2^S if it is found at level S in the tree—i.e., it has a side length of 2^S.

3. ALGORITHMS

We present four algorithms for obtaining the raster representation of an image given by a quadtree. The presentation is evolutionary in the sense that the second algorithm results from making some modifications to the first algorithm. The remaining algorithms are obtained in a similar manner.

Each algorithm traverses the quadtree in a row by row order. For each row in the image, every BLACK or WHITE node corresponding to a block which intersects the row is visited from left to right (e.g., for pixels 1–8 in the first row of Fig. 1a, node A of Fig. 1c is visited first followed by nodes B, C, D, and E). Each BLACK and WHITE node at level L in the tree is visited 2^L times (i.e., its height in pixels). The results of each such visit is that a run of length 2^L is output (e.g., a run of length 2 for node A of Fig. 1b).

The first algorithm has ALG1 as its main procedure. It is invoked with a pointer to the root of the quadtree representing the image and an integer corresponding to the log of the diameter of the image (e.g., n for a 2^n by 2^n image array). ALG1 outputs the image one row at a time by visiting in sequence, from left to right, every BLACK or WHITE node corresponding to a block which intersects the row. It is distinctive from the remaining algorithms, called ALG2, ALG3, and ALG4, respectively, in that each visit starts at the root of the subtree; whereas in ALG2 only visits to the leftmost block in each row start at the root; in ALG3 only visits to the first row of each leftmost block start at the root; and in ALG4 the only visit that starts at the root is the visit to the NW-most block in the image.

Procedure FINDBLOCK is used to locate the block containing the segment of the row that is currently being output. Its parameters are the coordinates of the leftmost raster point of the segment that is to be output and the coordinates of the lower right corner of a block (may be a nonterminal node) in the image containing the segment. FINDBLOCK partitions this block recursively until the smallest block corresponding to a terminal node is found that contains this segment. Procedure GETQUADRANT indicates which particular quadrant of a GRAY node contains a block corresponding to the segment which is to be output. For example, in locating the block containing the segment starting at row 0 and column 0 in Fig. 1, FINDBLOCK is successively invoked with blocks having lower right corners at $(7,7)$, $(3,3)$, and $(1,1)$. The result is block A.

All of our algorithms make use of procedures OUTPUTRUN and OUTPUTENDOFROW to do the actual output of the runs. For each BLACK or WHITE node of width W that participates in a row, OUTPUTRUN outputs a run of length W of the appropriate color (e.g., a BLACK run of length 2 for node A and row 1 of Fig. 1). OUTPUTENDOFROW outputs a separator symbol to mark the end of the row.

As an example of the application of ALG1, consider the image and quadtree given in Fig. 1. Nodes Ri correspond to nonterminal nodes. The terminal nodes in Fig. 1b have been labeled in the order in which they were visited for the first time. The result of the algorithm is the string W332, W242, W242, W17, W17, W17, W314, W8 where the comma serves as a separator symbol denoting the end of a row. The term W332 indicates that the first run is of length 3 and corresponds to WHITE, the second run is of length 3 and corresponds to BLACK, and the third run is of length 2 and corresponds to WHITE. When outputting the first row we start at node $R0$ and successively visit nodes $R1$ and A; $R1$, $R5$, B; $R1$, $R5$, C; $R2$, D; $R2$, E. For the second row we visit $R1$, A; $R1$, $R5$, F; $R1$, $R5$, G; $R2$, D; $R2$, E. For the third row we visit $R1$, $R6$, H; etc. The reader can verify that we visit a total of 104 nodes during this process.

ALG1 visits each segment of each row by repeatedly starting at the root of the quadtree. The second algorithm, having ALG2 as its main procedure, attempts to avoid this by using the structure of the tree to locate the immediately adjacent block to the last. For example, in Fig. 1, once the run corresponding to block B in the first row has been output, the next block to be visited is C. It can be located by traversing links corresponding to nodes $R5$ and C. This is in contrast to having to traverse links corresponding to nodes $R1$, $R5$, and C as is necessary when ALG1 is used. This is very much like an inorder traversal of the segment of the tree in which the row participates.

Adjacent nodes are located by use of a combination of procedures GTEQUAL_ADJ_NEIGHBOR and FINDBLOCK. GTEQUAL_ADJ_NEIGHBOR locates the smallest neighbor (recall the definition of neighbor in Section 2) block in a specified direction. If the node whose neighbor is sought is on the edge of the image and no neighbor exists in the direction searched, then NULL is returned. This signals that output for the row is finished (e.g., the eastern neighbor of node E in Fig. 1). If the neighboring block does exist, then a pointer to its corresponding node is returned. If it is a GRAY node (e.g., the eastern neighbor of node A in Fig. 1), then procedure FINDBLOCK is used to determine the adjacent BLACK or WHITE block which intersects the row currently being processed (e.g., the eastern adjacent block of A in Fig. 1 is B for the first row and F for the second row). Note that FINDBLOCK in this case is searching for the appropriate block which is on the extreme left edge of the GRAY block which it is currently partitioning and hence it is invoked with a value of 0 for the x-coordinate of the desired segment. The actual traversal of the blocks in which each row participates is controlled by procedure OUTROW.

Application of ALG2 to the image and quadtree given in Fig. 1 results in the same output string. However, the order in which nodes are visited is different. When outputting the first row we start at node $R0$ and successively visit nodes $R1$ and A; $R1$, $R5$, B; $R5$, C; $R5$, $R1$, $R0$, $R2$, D; $R2$, E; $R2$, $R0$. The last pair of nodes result in an indication that no neighbor exists—i.e., we have reached the end of the row. For the second row we visit $R1$, A; $R1$, $R5$, F; $R5$, G; $R5$, $R1$, $R0$, $R2$, D; $R2$, E; $R2$, $R0$. For the third row we visit $R1$, $R6$, H; etc. The reader can verify that we visit a total of 136 nodes during this process. Although for this particular example, ALG2 visits more nodes than ALG1, we shall see in Section 4 when ALG2 is superior to ALG1.

ALG2 visits the first segment of each row of the leftmost blocks by traversing links starting at the root of the tree. For example, in Fig. 1, when outputting the first and second rows, nodes $R1$ and A are visited twice—once for each row in which block A participates as the initial segment. ALG3 avoids this by starting the output of all but the first row in the block at the block. For example, in Fig. 1 once the first row has been output, i.e., nodes $R1$, A; $R1$, $R5$, B; $R5$, C; $R5$, $R1$, $R0$, $R2$, D; $R2$, E; and $R2$, $R0$ have been visited, we output the second row by starting at A and then visit $R1$, $R5$, F; $R5$, G; $R5$, $R1$, $R0$, $R2$, D; $R2$, E; $R2$, $R0$. For the third row we start at the root and locate H by going through $R1$, $R6$, H; etc. The reader can verify that we visit a total of 132 nodes during this process.

ALG3 visits the first segment of the first row of each of the leftmost blocks by repeatedly starting at the root of the quadtree. The fourth algorithm, having ALG4 as its main procedure, avoids this by using the structure of the tree to locate the immediately adjacent block to the south. For example in Fig. 1, once the rows having their first segment in block H have been output (i.e., the third row), the next row to be output has its first segment in block N (i.e., the fourth row). This block is located by traversing links corresponding to nodes $R6$ and N. This is in contrast to having to traverse links corresponding to nodes $R1$, $R6$, and N as is necessary when ALG1, ALG2, or ALG3 are used. In essence, we have applied the same principle of inorder traversal to transitions in the vertical direction as was applied to transitions in the horizontal direction in the transformation of ALG1 to ALG2.

Application of ALG4 to the image and quadtree given in Fig. 1 yields the same output string as was obtained for ALG1 and ALG2. However, the order in which nodes are visited is different. When outputting the first row we start at node *R*0 and successively visit nodes *R*1 and *A*; *R*1, *R*5, *B*; *R*5, *C*; *R*5, *R*1, *R*0, *R*2, *D*; *R*2, *E*; *R*2, *R*0. For the second row we start at *A* and then visit *R*1, *R*5, *F*; *R*5, *G*; *R*5, *R*1, *R*0, *R*2, *D*; *R*2, *E*; *R*2, *R*0. For the third row we again start at *A* except that now we must locate an adjacent block to the south. This takes us through nodes *R*1, *R*6, and *H*. We next visit *R*6, *I*; etc. The reader can verify that we visit a total of 136 nodes during this process. Although for Fig. 1 ALG4 and ALG2 both visit the same number of nodes, we shall see in Section 4 when ALG4 is superior to ALG2.

```
procedure ALG1 (ROOT, LEVEL);
/*Output a raster representation of the 2↑LEVEL by 2↑LEVEL image corre-
   sponding to the quadtree rooted at node ROOT. For each row each block
   containing a segment of the row is located by descending the appropriate links
   from ROOT*/
begin
  value node ROOT;
  value integer LEVEL;
  node P;
  integer DIAMETER, WIDTH, X, Y;
  DIAMETER ← 2↑LEVEL;
  for Y ← 0 step 1 until DIAMETER − 1 do
    begin /*Process the rows in sequence one row at a time */
      X ← 0; while X < DIAMETER do
        begin /*Process the blocks in each row from left to right*/
          P ← ROOT;
          WIDTH ← DIAMETER;
          FINDBLOCK(P, X, DIAMETER, Y, DIAMETER, WIDTH);
          OUTPUTRUN(NODETYPE(P), WIDTH);
          X ← X + WIDTH;
        end;
      OUTPUTENDOFROW( );
    end;
end;
```

```
procedure FINDBLOCK(P, X, XFAR, Y, YFAR, W);
/*P is the root of a block of width W having its lower right corner at (XFAR −
   1, YFAR − 1). If P is BLACK or WHITE, then return the values of P and W;
   otherwise, repeat the procedure for the son of P that has its upper left corner at
   (X, Y)*/
begin
  reference node P;
  value integer X, XFAR, Y, YFAR;
  reference integer W;
  quadrant Q;
  if GRAY(P) then
```

```
    begin
      W ← W/2;
      Q ← GETQUADRANT(X, XFAR − W, Y, YFAR − W);
      P ← SON(P, Q);
      case Q of
        'NW': FINDBLOCK(P, X, XFAR − W, Y, YFAR − W, W)
        'NE': FINDBLOCK(P, X, XFAR, Y, YFAR − W, W)
        'SW': FINDBLOCK(P, X, XFAR − W, Y, YFAR, W)
        'SE': FINDBLOCK(P, X, XFAR, Y, YFAR, W)
      end;
    end;
end;


quadrant procedure GETQUADRANT(X, XCENTER, Y, YCENTER);
/* Find the quadrant of the block rooted at (XCENTER, YCENTER) that contains
   (X, Y) */
begin
  value integer X, XCENTER, Y, YCENTER;
  return (if X < XCENTER then
              if Y < YCENTER then 'NW'
              else 'SW'
          else if Y < YCENTER then 'NE'
          else 'SE');
end;


procedure ALG2 (ROOT, LEVEL);
/* Output a raster representation of the 2↑LEVEL by 2↑LEVEL image corre-
   sponding to the quadtree rooted at node ROOT. For each row, the leftmost block
   is found and then the blocks comprising the row are visited in sequence by
   ascending and descending the appropriate links in the tree */
begin
  value node ROOT;
  value integer LEVEL;
  node P;
  integer DIAMETER, WIDTH, Y;
  DIAMETER ← 2↑LEVEL;
  for Y ← 0 step 1 until DIAMETER − 1 do
    begin /* Process the rows in sequence one row at a time */
      P ← ROOT;
      WIDTH ← DIAMETER;
      FINDBLOCK(P, 0, DIAMETER, Y, DIAMETER, WIDTH);
      /* Find the leftmost block containing row Y */
      OUTROW(P, Y, LOG2(WIDTH));
      /* LOG2 returns the log of WIDTH to base 2 */
    end;
end;
```

```
procedure OUTROW (P, ROW, L);
/*Output a raster corresponding to all of the blocks that have segments in row
   ROW starting with node P at level L*/
begin
  value node P;
  value integer L, ROW;
  node Q;
  integer WIDTH;
  WIDTH ← 2↑L;
  do
    begin
      OUTPUTRUN(NODETYPE(P), WIDTH);
      /*Find the leftmost adjacent block containing row ROW*/
      GTEQUAL_ADJ_NEIGHBOR(P, 'E', Q, L);
      WIDTH ← 2↑L;
      if GRAY(Q) then
        FINDBLOCK(Q, 0, WIDTH, ROW,
                    ROW + WIDTH − (ROW MOD WIDTH), WIDTH);
     P ← Q;
    end
    until NULL(P);
end;

procedure GTEQUAL_ADJ_NEIGHBOR(P, D, Q, L);
/*Return in Q the neighbor of node P in horizontal or vertical direction D. L
   denotes the level of the tree at which node P is initially found and the level of the
   tree at which node Q is ultimately found. If such a node does not exist, then
   return NULL*/
begin
  value node P;
  value direction D;
  reference node Q;
  reference integer L;
  L ← L + 1;
  if not NULL (FATHER(P)) and ADJ(P, SONTYPE(P)) then
    /*Find a common ancestor*/
    GTEQUAL_ADJ_NEIGHBOR(FATHER(P), D, Q, L)
  else Q ← FATHER(P);
  /*Follow the reflected path to locate the neighbor*/
  if not NULL (Q) and GRAY(Q) then
    begin
      Q ← SON(Q, REFLECT(D, SONTYPE(P)));
      L ← L − 1;
    end;
end;

procedure ALG3 (ROOT, LEVEL);
/*Output a raster representation of the 2↑LEVEL by 2↑LEVEL image corre-
```

```
    sponding to the quadtree rooted at node ROOT. For each row, the leftmost block
    is found and then the blocks comprising the row are visited in sequence by
    ascending and descending the appropriate links in the tree. ALG3 differs from
    ALG2 in that when a block contains the initial segment of more than one row, it
    is not repeatedly searched for when outputting the remaining rows in which it
    participates*/
begin
  value node ROOT;
  value integer LEVEL;
  node P;
  integer DIAMETER, ROW, WIDTH, Y;
  DIAMETER ← 2↑LEVEL;
  Y ← 0;
  while Y < DIAMETER do
    begin /*Process the leftmost blocks in sequence one block at a time*/
      P ← ROOT;
      WIDTH ← DIAMETER;
      FINDBLOCK(P, 0, DIAMETER, Y, DIAMETER, WIDTH);
      /*Find the leftmost block containing row Y*/
      for ROW ← Y step 1 until Y + WIDTH − 1 do
          OUTROW(P, ROW, LOG2(WIDTH));
          /*LOG2 returns the log of WIDTH to base 2*/
      Y ← Y + WIDTH;
    end;
end;

procedure ALG4 (ROOT, LEVEL);
/*Output a raster representation of the 2↑LEVEL by 2↑LEVEL image corre-
    sponding to the quadtree rooted at node ROOT. For each row, the leftmost block
    is located and then the blocks comprising the row are visited in sequence by
    ascending and descending the appropriate links in the tree. Successive blocks in
    the vertical direction are located in the same manner*/
begin
  value node ROOT;
  value integer LEVEL;
  node P, Q;
  integer ROW, WIDTH, Y;
  P ← ROOT;
  WIDTH ← 2↑LEVEL;
  FINDBLOCK(P, 0, WIDTH, 0, WIDTH, WIDTH);
  /*Find the leftmost block containing row 0*/
  Y ← 0;
  do
    begin
      for ROW ← Y step 1 until Y + WIDTH − 1 do
          OUTROW(P, ROW, LOG2(WIDTH));
          /*LOG2 returns the log of WIDTH to base 2*/
      Y ← Y + WIDTH;
```

```
        /* Find the leftmost block containing row Y */
        GTEQUAL_ADJ_NEIGHBOR(P,'S', Q, LEVEL);
        WIDTH ← 2↑LEVEL;
        if GRAY(Q) then FINDBLOCK(Q,0, WIDTH, Y, Y + WIDTH, WIDTH);
        P ← Q;
     end
     until NULL(P);
end;
```

4. ANALYSIS

We analyze the various quadtree-to-raster algorithms in the order in which they were presented (i.e., ALG1, ALG2, ALG3, and ALG4). As transitions are made between the algorithms we show how the modifications affect our time estimates. We measure the running time of the various algorithms by counting the nodes that they visit. Thus we are really only concerned with the time spent by procedure FINDBLOCK and GTEQUAL_ADJ_NEIGHBOR. Note that in all our analyses we count a visit to node B from node A where there is a link from A to B as a one node visit.

THEOREM 1. *Given a 2^n by 2^n image with b_i blocks of size 2^i, the number of nodes visited by* ALG1 *is* $\Sigma_{i=0}^{n-1}(n-i)\cdot b_i\cdot 2^i$.

Proof. Observe that for each block of size 2^i there are 2^i rows each of which is visited once starting at the root of the quadtree. But each block of size 2^i is at a distance of $n-i$ nodes from the root and our result follows. Q.E.D.

Theorem 1 is interesting because it means that for any image the number of nodes that are visited by ALG1 in outputting it only depends on the number of blocks comprising it and their respective sizes. The relative position of the blocks is irrelevant. Thus different images are seen to require the same number of node visits. For example, the image in Fig. 3 requires the same number of node visits as the image in Fig. 1. Intuitively, this is not surprising because ALG1 processes the rows and the segments of blocks comprising them independently of one another.

The analysis of ALG2 is more complex. We first make the following observation. ALG2 as well as ALG3 and ALG4 visit the various segments of each row by exploiting the tree structure to find neighboring blocks. In doing so, the number of horizontal adjacencies between blocks that are explored is equal to the sum of the

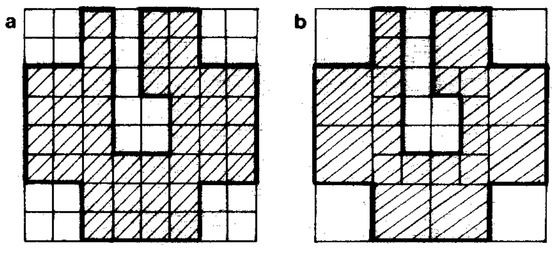

FIG. 3. An image and its maximal blocks which requires the same number of nodes to be visited by ALG1, ALG2, and ALG3 as does Fig. 1. (a) Sample image. (b) Block decomposition of the image in (a).

heights of the blocks comprising the image. This is true because each block will be visited once for each row in which it is a member. Theorem 2 will show that the number of nodes visited by ALG2 is bounded by four times this number. However, we must first prove the following lemma. Assume a 2^n by 2^n image. If the image is a complete quadtree, i.e., all blocks in the image are at level 0, then we have:

LEMMA 1. *In a complete quadtree, the number of nodes visited by* ALG2 *is bounded by four times the number of blocks in the image (more precisely, it is equal to four times the difference between the area and the diameter of the image, where the diameter is 2^n for a 2^n by 2^n image).*

Proof. Starting at the root node of the quadtree, for each row in a 2^n by 2^n image, n nodes are visited when locating the node corresponding to the leftmost block. Once this is done, GTEQUAL_ADJ_NEIGHBOR is invoked 2^n times to find neighbors in the eastern direction. Of the nodes corresponding to blocks in the row, 2^0 have their nearest common ancestor at level n, 2^1 at level $n-1, \ldots, 2^i$ at level $n-i$, and 2^{n-1} at level 1. Once the nearest common ancestor has been found, a path of equal length must be traversed to locate the adjacent neighbors. In addition, the node corresponding to the rightmost block in each row has no eastern neighbor. This is detected by attempting to locate a nonexistent common ancestor—a process which traverses a path of length n (i.e., to the root of the quadtree and including it). Since there are 2^n rows, the number of nodes visited is

$$\begin{aligned} 2^n \cdot \left(n + 2 \cdot \sum_{i=1}^{n} i \cdot 2^{n-i} + n\right) &= 2^n \cdot \left(2n + 2^{n+1} \cdot \sum_{i=1}^{n} \frac{i}{2^i}\right) \\ &= 2^n \cdot \left(2n + 2^{n+1} \cdot \left(2 - \frac{n+2}{2^n}\right)\right) \\ &= 2^{2n+2} - 2^{n+2}. \end{aligned}$$

However, there are 2^{2n} blocks in the image. Thus the number of nodes that have been visited is bounded by four times the number of blocks in the image. Q.E.D.

Another way of phrasing this result is that the average distance between two adjacent quadtree nodes is bounded by 4. The same result is shown in [1] for the representation of two-dimensional arrays as binary trees. However, the bound is obtained by techniques that make use of recurrence relations.

We are now ready to prove the main result.

THEOREM 2. *For any image, the number of nodes visited by* ALG2 *is bounded by four times the sum of the heights of the blocks in the image, i.e., $4 \cdot \sum_{i=0}^{n} b_i \cdot 2^i$ for a 2^n by 2^n image with b_i blocks of size 2^i.*

Proof. By Lemma 1 the theorem is true for a complete quadtree. We shall use induction on the size of the blocks to show the result for any quadtree.

Consider a 2 by 2 pixel block in the complete quadtree and assume that the four blocks corresponding to the pixels have been merged to yield one block. Since we are processing the image in a row-by-row manner, the only adjacencies that are eliminated by the merge are the horizontal ones between the blocks being merged (e.g., between 19 and 20 and 27 and 28 in Fig. 1a). This means that four less nodes will be visited by our algorithm. In addition, the node corresponding to the merged block (e.g., node J for the blocks corresponding to 19, 20, 27, and 28 in Figs. 1a and b) is one node closer to its horizontal neighbors to the left (e.g., blocks 18 and 26 in Fig. 1a) and right (e.g., blocks 21 and 29 in Fig. 1a). Thus we find that $4 + 1 + 1 + 1 + 1 = 8$ less nodes will be visited. However, the total height of the blocks in the image has decreased by 2 (initially there were 4 blocks of size 1 and now there is one block of size 2) and our theorem holds.

More generally, consider a 2^{S+1} by 2^{S+1} block, i.e., we are merging four 2^S by 2^S blocks. Once again, since we are processing the image in a row-by-row manner, the only adjacencies that are eliminated by the merge are the horizontal ones between the blocks being merged. Since the blocks are of size 2^S, 2^{S+1} adjacencies are eliminated. Moreover, each block of size 2^S is at a distance of 2 from its horizontal neighbor with which it is being merged. Thus the elimination of 2^{S+1} adjacencies results in $2 \cdot 2^{S+1}$ less nodes being visited by the algorithm. In addition, the node corresponding to the merged block is one node closer to each of its horizontal neighbors to the left and right. However, there are 2^{S+1} neighbors in each of the left and right directions. Thus the total number of nodes that will be visited has decreased by $4 \cdot 2^{S+1}$. But the total height of the blocks in the image has decreased by 2^{S+1} (initially there were four blocks of size 2^S and now there is one block of size 2^{S+1}) and our theorem holds. Q.E.D.

Theorem 2 is useful in comparing ALG1 and ALG2. We see from Theorem 2 that for a 2^n by 2^n image with b_i blocks of size 2^i the number of nodes that are visited by ALG2 is bounded by $4 \cdot \Sigma_{i=0}^{n} b_i \cdot 2^i$ in contrast with $\Sigma_{i=0}^{n}(n-i) \cdot b_i \cdot 2^i$ for ALG1. Thus there is less dependence on the resolution of the image (i.e., n) when ALG2 is used. In particular, for $n \geq 4$ ALG2 is potentially superior to ALG1 as can be seen for a complete quadtree for $n = 4$. In fact, as n gets large, the majority of the nodes appear deeper in the tree (i.e., at a lower level). Since in such a case $(n - i) \cdot b_i \cdot 2^i > 4 \cdot b_i \cdot 2^i$ we find that ALG2 will be more efficient than ALG1.

Also note that a large number of nodes in ALG2 are visited through the use of GTEQUAL_ADJ_NEIGHBOR rather than FINDBLOCK. This leads to even greater efficiency since with GTEQUAL_ADJ_NEIGHBOR a decision as to which node to visit next (i.e., the link to be traversed) only depends on a table lookup operation (using the ADJ and REFLECT relationships) while FINDBLOCK requires a certain amount of arithmetic to be done. Note that for the sake of clarity, our exposition of algorithms ALG1 and ALG2 and their interactions with procedures FINDBLOCK and GTEQUAL_ADJ_NEIGHBOR leads to less than optimal code (e.g., global variables could eliminate the need for many of the formal parameters and recursion could be replaced by loops). In fact, FINDBLOCK is not necessary in ALG2. Instead, we could make use of a procedure which only checks for a NW or SW son. For example, in Fig. 1, when processing the third row and searching for an eastern neighbor of node K, application of GTEQUAL_ADJ_NEIGHBOR yields node $R7$. Subsequent application of FINDBLOCK yields node L. However, we know that the NE and SE links are impossible. Thus checking for them in FINDBLOCK and GETQUADRANT is unnecessary.

The construction used in the proof of Theorem 2 is worthy of further attention. It can be used in conjunction with the result of Lemma 1 to compute exactly how many nodes will be visited for any image given the number of blocks comprising it and their respective sizes. Thus, as was the case for ALG1, different images will require the same number of node visits when ALG2 is used. We have the following theorem:

THEOREM 3. *Given a 2^n by 2^n image with b_i blocks of size 2^i, the number of nodes visited by* ALG2 *is*

$$2^{2n+2} - 2^{n+2} - \sum_{i=1}^{n} b_i \cdot (2^{2i+2} - 2^{i+2}).$$

Proof. From the proof of Lemma 1 we have that traversing a complete quadtree of size 2^i requires $2^{2i+2} - 2^{i+2}$ nodes to be visited. This represents a traversal starting and terminating at a common ancestor. Since the 2^i by 2^i array of pixels has been replaced by one block, $2^{2i+2} - 2^{i+2}$ less nodes will be visited for a block of size 2^i. Recall that if the array contains 2^n by 2^n blocks of size 1, then $2^{2n+2} - 2^{n+2}$ nodes will be visited. Subtracting the contribution of b_i blocks of size 2^i yields the desired result. Q.E.D.

The analysis of ALG3 is relatively simple. Recall that the only difference between ALG2 and ALG3 is that the leftmost blocks are not repeatedly located by starting at the root for each row in which they participate. The following theorem summarizes the execution time of ALG3.

THEOREM 4. *Given a 2^n by 2^n image with b_i blocks of size 2^i where v_i blocks of size 2^i have segments that include the first column, the number of nodes visited by* ALG3 *is*

$$2^{2n+2} - 2^{n+2} - \sum_{i=1}^{n} b_i \cdot (2^{2i+2} - 2^{i+2}) - \sum_{i=0}^{n} (n-i) \cdot v_i \cdot (2^i - 1).$$

Proof. Using ALG2, v_i blocks of size 2^i in the first column lead to $(n-i) \cdot v_i \cdot 2^i$ nodes being visited. Using ALG3, only $(n-i)$ nodes are visited for each block of size 2^i in the first column. Combining these facts with Theorem 3 leads to the desired result. Q.E.D.

Clearly ALG3 is superior to ALG2 since it does not repeatedly traverse a path from the root to each row of a block in the first column. Thus ALG2 should never be used. However, the analysis of ALG2 is useful in the analysis of ALG4 since ALG4 is related to ALG2 in the same way as ALG2 is related to ALG1. To see this, recall that ALG2 used neighbor finding techniques to locate adjacent blocks when processing a row while ALG4 uses neighbor finding techniques to locate adjacent blocks in the vertical direction as well. In order to obtain the number of nodes visited by ALG4 we first prove the following lemma:

LEMMA 2. *Given a 2^n by 2^n image with v_i blocks of size 2^i in the first column, the number of nodes visited by* ALG4 *in locating the nodes corresponding to the first blocks in each column is*

$$2^{n+2} - 4 - \sum_{i=1}^{n} v_i \cdot (2^{i+2} - 4).$$

Proof. From Lemma 1 we have that traversing a row of size 2^i pixels in a 2^i by 2^i image requires $2^{i+2} - 4$ nodes to be visited. This represents a traversal starting and

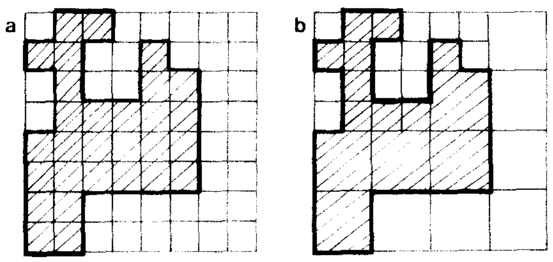

FIG. 4. An image and its maximal blocks which requires the same number of nodes to be visited by ALG4 as does Fig. 1. (a) Sample image. (b) Block decomposition of the image in (a).

terminating at a common ancestor. Since the 2^i pixels have been replaced by a segment of one block, $2^{i+2} - 4$ fewer nodes will be visited for a block of size 2^i. Recall that if the row contains 2^n blocks of size 1, then $2^{n+2} - 4$ nodes will be visited. Subtracting the contribution of v_i blocks of size 2^i yields the desired result. Q.E.D.

THEOREM 5. *Given a 2^n by 2^n image with b_i blocks of size 2^i, where v_i blocks of size 2^i have segments that include the first column, the number of nodes visited by* ALG4 *is*

$$2^{2n+2} - (n+4)\cdot 2^n - 4 - \sum_{i=1}^{n} b_i \cdot (2^{2i+2} - 2^{i+2}) + \sum_{i=0}^{n} v_i \cdot (i \cdot 2^i + 4).$$

Proof. We use the result of Theorem 3 in conjunction with the fact that instead of the first column leading to $\sum_{i=0}^{n}(n-i)\cdot v_i \cdot 2^i$ nodes being visited, only $2^{n+2} - 4 - \sum_{i=1}^{n} v_i \cdot (2^{i+2} - 4)$ nodes are actually visited. The identity $\sum_{i=0}^{n} v_i \cdot 2^i = 2^n$ leads to the desired result. Q.E.D.

Theorem 5 shows that ALG4 also has a degree of configuration independence. However, the difference between it and ALG1 is that the first columns of the different images must participate in the same number of blocks of the different sizes. Thus Figs. 1 and 3 do not result in the same number of nodes being visited whereas Figs. 1 and 4 do.

5. CONCLUDING REMARKS

We have presented a number of algorithms for converting a quadtree representation of a binary image to a raster representation of the image. Our presentation was an evolutionary one so that we could analyze their execution times and see their relative advantages and disadvantages. The running time of all of the algorithms was shown, in essence, to be proportional to the sum of the heights of the blocks comprising the image. In other words, the amount of work required is directly proportional to the complexity of the image; i.e., two different images will require the same amount of work if they have the same number of blocks of each size. This is not surprising when we recall the row-by-row nature of our algorithms.

Of the algorithms that we presented, ALG1 is the most straightforward. Recall that it proceeds in a top-down manner while ALG2, ALG3, and ALG4 proceed in a bottom-up manner. We stated earlier that ALG2 is potentially superior to ALG1 when the resolution of the image increases ($n \geq 4$). As an example consider the complete quadtree corresponding to a 2^4 by 2^4 image. This is because the execution time of ALG2 is always bounded by four times the sum of the heights of the blocks in the image. ALG3 has the same relationship to ALG1 as does ALG2 except that ALG3 is always superior to ALG2. ALG4 makes use of the bottom-up methods of tree traversal for making transitions in the vertical direction as well, and thus it is potentially superior to ALG3 when the resolution increases ($n \geq 4$) and there is a relatively large number of blocks in the first column at the lower levels of the tree. Note that Theorem 2 does not hold for ALG4 and thus it is not always true that the execution time of ALG4 is bounded by four times the heights of the blocks in the image. For example, see Fig. 5 where there are eight blocks of size 1 in the first column. In this case ALG4 results in 140 nodes being visited while ALG2 and ALG3 only visit 136 nodes.

As mentioned earlier, our algorithms have been presented in a way that enhances their clarity. Greater efficiency could be achieved by using global variables in procedure FINDBLOCK and replacing instances of recursion by loops. Also, there are a number of cases where a call to FINDBLOCK is not necessary. Instead, links in a predetermined direction can be traversed. For example, in ALG4, when a southern neighbor is GRAY, the call to FINDBLOCK can be replaced by traversing NW links until a terminal node is reached (e.g., in Fig. 1, the desired adjacent block to the south of A is H which can be reached by traversing the NW links of A's southern neighbor $R6$).

It should be clear that we have not exhausted the possible algorithms for achieving our goal of outputting the raster representation of a quadtree. An alternative approach would avoid the work required by GTEQUAL_ADJ_NEIGHBOR and FINDBLOCK in locating neighboring nodes in each row in the image by linking such nodes. This is similar to the concept of roping [4–6]. The disadvantage of such a technique is the amount of extra space required to store the links. In addition, as we have shown, the cost of our neighbor finding techniques is not very high (i.e., four nodes must be visited per adjacency rather than one as is the case when nodes are linked). Nevertheless, this does suggest another variation on our algorithms as discussed below.

We can maintain a linked list of all the blocks which participate in a row. As we process each run in a row, we check if the particular run is the last row in which the block participates. If this is the case then we delink the block from the linked list and replace it with its adjacent neighbor in the S direction. If this neighbor is a GRAY node, then we replace it with all the northernmost descendants of the GRAY

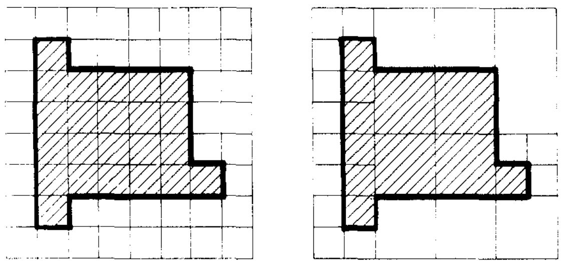
FIG. 5. An image and its maximal blocks for which ALG4 visits more nodes than ALG2 or ALG3.

node (e.g., T in Fig. 1 has $R9$ as its S neighbor and since $R9$ is GRAY, T is replaced by Z and AA). The advantage of this method is that we need not look for neighbors in the E direction except when the linked list corresponding to the first row is built. Of course, we now look for neighbors in the S direction instead. However, this is done once per block rather than for each row in which the block participates. It can be shown in a manner similar to the proof of Theorem 2 that the execution time of this algorithm is bounded by $\sum_{i=0}^{n} b_i \cdot 2^i + 4 \cdot \sum_{i=0}^{n} b_i$ instead of $4 \cdot \sum_{i=0}^{n} b_i \cdot 2^i$. Note that the actual execution time of this variation will have higher constants of proportionality due to the need for queue manipulation. Also, in the case of complete quadtrees the earlier methods are superior.

REFERENCES

1. R. A. DeMillo, S. C. Eisenstat, and R. J. Lipton, Preserving average proximity in arrays, *Comm. ACM*, March 1978, 228–231.
2. C. R. Dyer, A. Rosenfeld, and H. Samet, Region representation: Boundary codes from quadtrees, *Comm. ACM*, March 1980, 171–179.
3. R. A. Finkel and J. L. Bentley, Quad trees: A data structure for retrieval on composite keys, *Acta Inform.* **4**, 1974, 1–9.
4. G. M. Hunter, Efficient Computation and Data Structures for Graphics, Ph.D. dissertation, Department of Electrical Engineering and Computer Science, Princeton University, Princeton, N.J., 1978.
5. G. M. Hunter and K. Steiglitz, Operations on images using quadtrees, *IEEE Trans. Pattern Anal. Mach. Intell.* **1**, 1979, 145–153.
6. G. M. Hunter and K. Steiglitz, Linear transformation of pictures represented by quadtrees, *Comput. Graphics Image Processing* **10**, 1979, 289–296.
7. C. L. Jackins and S. L. Tanimoto, Oct-trees and their use in representing three-dimensional objects, *Comput. Graphics Image Processing* **14**, 1980, 249–270.
8. A. Klinger, Patterns and search statistics, in *Optimizing Methods in Statistics* (J. S. Rustagi, Ed.), Academic Press, New York, 1971.
9. A. Klinger and C. R. Dyer, Experiments on picture representation using regular decomposition, *Comput. Graphics Image Processing* **5**, 1976, 68–105.
10. A. Klinger and M. L. Rhodes, Organization and access of image data by areas, *IEEE Trans. Pattern Anal. Mach. Intell.* **1**, 1979, 50–60.
11. D. E. Knuth, *The Art of Computer Programming*, Vol 1, *Fundamental Algorithms*, 2nd ed., Addison-Wesley, Mass., 1973.
12. P. Naur (Ed.), Revised report on the algorithmic language ALGOL 60, *Comm. ACM* **3**, 1960, 299–314.
13. H. Samet, Region representation: Quadtrees from boundary codes, *Comm. ACM*, March 1980, 163–170.
14. H. Samet, Region representation: Quadtrees from binary arrays, *Comput. Graphics Image Processing* **13**, 1980, 88–93.
15. H. Samet, An algorithm for converting rasters to quadtrees, *IEEE Trans. Pattern Anal. Mach. Intell.* **3**, 1981, 93–95.